L 49

49.1 LAHF—Load Status Flags into AH Register

Opcode	Instruction	Description
9F	LAHF	Load: AH = EFLAGS(SF:ZF:0:AF:0:PF:1:CF)

Description

Moves the low byte of the EFLAGS register (which includes status flags SF, ZF, AF, PF, and CF) to the AH register. Reserved bits 1, 3, and 5 of the EFLAGS register are set in the AH register as shown in the "Operation" section below.

Operation

```
AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
```

Flags Affected

None (that is, the state of the flags in the EFLAGS register are not affected).

Exceptions (All Operating Modes)

None.

49.2 LAR—Load Access Rights Byte

Opcode	Instruction	Description
0F 02 /*r*	LAR *r16,r/m16*	*r16* ← *r/m16* masked by FF00H
0F 02 /*r*	LAR *r32,r/m32*	*r32* ← *r/m32* masked by 00FxFF00H

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

When the operand size is 32 bits, the access rights for a segment descriptor include the type and DPL fields and the S, P, AVL, D/B, and G flags, all of which are located in the second doubleword (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FXFF00H before it

is loaded into the destination operand. When the operand size is 16 bits, the access rights include the type and DPL fields. Here, the two lower-order bytes of the doubleword are masked by FF00H before being loaded into the destination operand.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode.

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	Yes
5	16-bit/32-bit task gate	Yes
6	16-bit interrupt gate	No
7	16-bit trap gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	Yes
D	Reserved	No
E	32-bit interrupt gate	No
F	32-bit trap gate	No

Operation

```
IF SRC(Offset) > descriptor table limit THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
    AND (CPL > DPL) OR (RPL > DPL)
    OR Segment type is not valid for instruction
        THEN
            ZF ← 0
        ELSE
            IF OperandSize = 32
```

```
                THEN
                    DEST ← [SRC] AND 00FxFF00H;
                ELSE (*OperandSize = 16*)
                    DEST ← [SRC] AND FF00H;
            FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

Real-Address Mode Exceptions

#UD	The LAR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.

49.3 LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Description
C5 /*r*	LDS *r16,m16:16*	Load DS:*r16* with far pointer from memory
C5 /*r*	LDS *r32,m16:32*	Load DS:*r32* with far pointer from memory
0F B2 /*r*	LSS *r16,m16:16*	Load SS:*r16* with far pointer from memory
0F B2 /*r*	LSS *r32,m16:32*	Load SS:*r32* with far pointer from memory
C4 /*r*	LES *r16,m16:16*	Load ES:*r16* with far pointer from memory
C4 /*r*	LES *r32,m16:32*	Load ES:*r32* with far pointer from memory
0F B4 /*r*	LFS *r16,m16:16*	Load FS:*r16* with far pointer from memory
0F B4 /*r*	LFS *r32,m16:32*	Load FS:*r32* with far pointer from memory
0F B5 /*r*	LGS *r16,m16:16*	Load GS:*r16* with far pointer from memory
0F B5 /*r*	LGS *r32,m16:32*	Load GS:*r32* with far pointer from memory

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits

or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a null selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

Operation

```
IF ProtectedMode
    THEN IF SS is loaded
        THEN IF SegementSelector = null
            THEN #GP(0);
        FI;
        ELSE IF Segment selector index is not within descriptor table limits
        OR Segment selector RPL ≠ CPL
        OR Access rights indicate nonwritable data segment
        OR DPL ≠ CPL
            THEN #GP(selector);
        FI;
        ELSE IF Segment marked not present
            THEN #SS(selector);
        FI;
        SS ← SegmentSelector(SRC);
        SS ← SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS, or GS is loaded with non-null segment selector
        THEN IF Segment selector index is not within descriptor table limits
        OR Access rights indicate segment neither data nor readable code segment
        OR (Segment is data or nonconforming-code segment
            AND both RPL and CPL > DPL)
            THEN #GP(selector);
        FI;
        ELSE IF Segment marked not present
            THEN #NP(selector);
        FI;
        SegmentRegister ← SegmentSelector(SRC) AND RPL;
        SegmentRegister ← SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS or GS is loaded with a null selector:
        SegmentRegister ← NullSelector;
        SegmentRegister(DescriptorValidBit) ← 0; (*hidden flag; not accessible by software*)
    FI;
FI;
IF (Real-Address or Virtual-8086 Mode)
    THEN
        SegmentRegister ← SegmentSelector(SRC);
FI;
DEST ← Offset(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If a null selector is loaded into the SS register.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL. If the DS, ES, FS, or GS register is being loaded with a non-null segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-null segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If source operand is not a memory location.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

49.4 LEA—Load Effective Address

Opcode	Instruction	Description
8D /*r*	LEA *r16,m*	Store effective address for *m* in register *r16*
8D /*r*	LEA *r32,m*	Store effective address for *m* in register *r32*

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

Operation

```
IF OperandSize = 16 AND AddressSize = 16
    THEN
        DEST ← EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 AND AddressSize = 32
        THEN
            temp ← EffectiveAddress(SRC); (* 32-bit address *)
            DEST ← temp[0..15]; (* 16-bit address *)
    ELSE IF OperandSize = 32 AND AddressSize = 16
        THEN
            temp ← EffectiveAddress(SRC); (* 16-bit address *)
            DEST ← ZeroExtend(temp); (* 32-bit address *)
    ELSE IF OperandSize = 32 AND AddressSize = 32
        THEN
            DEST ← EffectiveAddress(SRC); (* 32-bit address *)
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD If source operand is not a memory location.

Real-Address Mode Exceptions

#UD If source operand is not a memory location.

Virtual-8086 Mode Exceptions

#UD If source operand is not a memory location.

49.5 LEAVE—High Level Procedure Exit

Opcode	Instruction	Description
C9	LEAVE	Set SP to BP, then pop BP
C9	LEAVE	Set ESP to EBP, then pop EBP

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the *Intel Architecture Software Developer's Manual, Volume 3*, for detailed information on the use of the ENTER and LEAVE instructions.

Operation

```
IF StackAddressSize = 32
    THEN
        ESP ← EBP;
    ELSE (* StackAddressSize = 16*)
        SP ← BP;
FI;
IF OperandSize = 32
    THEN
        EBP ← Pop();
    ELSE (* OperandSize = 16*)
        BP ← Pop();
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the EBP register points to a location that is not within the limits of the current stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If the EBP register points to a location outside of the effective address space from 0 to 0FFFFH.

Virtual-8086 Mode Exceptions

#GP(0)	If the EBP register points to a location outside of the effective address space from 0 to 0FFFFH.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

49.6 LES—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

49.7 LFS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

49.8 LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /2	LGDT *m16&32*	Load *m* into GDTR
0F 01 /3	LIDT *m16&32*	Load *m* into IDTR

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

See “SGDT/SIDT—Store Global/Interrupt Descriptor Table Register” in this chapter for information on storing the contents of the GDTR and IDTR.

Operation

```
IF instruction is LIDT
    THEN
        IF OperandSize = 16
```

```
            THEN
                IDTR(Limit) ← SRC[0:15];
                IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
            ELSE (* 32-bit Operand Size *)
                IDTR(Limit) ← SRC[0:15];
                IDTR(Base) ← SRC[16:47];
        FI;
    ELSE (* instruction is LGDT *)
        IF OperandSize = 16
            THEN
                GDTR(Limit) ← SRC[0:15];
                GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
            ELSE (* 32-bit Operand Size *)
                GDTR(Limit) ← SRC[0:15];
                GDTR(Base) ← SRC[16:47];
        FI; FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If source operand is not a memory location.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

49.9 LGS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

49.10 LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /2	LLDT *r/m16*	Load segment selector *r/m16* into LDTR

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses to segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If the source operand is 0, the LDTR is marked invalid and all references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. Also, this instruction can only be executed in protected mode.

Operation

```
IF SRC(Offset) > descriptor table limit THEN #GP(segment selector); FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ LDT THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
LDTR(SegmentSelector) ← SRC;
LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The LLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LLDT instruction is recognized in virtual-8086 mode.

49.11 LIDT—Load Interrupt Descriptor Table Register

See entry for LGDT/LIDT—Load Global/Interrupt Descriptor Table Register.

49.12 LMSW—Load Machine Status Word

Opcode	Instruction	Description
0F 01 /6	LMSW *r/m16*	Loads *r/m16* in machine status word of CR0

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on the Pentium Pro, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

Operation

```
CR0[0:3] ← SRC[0:3];
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

49.13 LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Description
F0	LOCK	Asserts LOCK# signal for duration of the accompanying instruction

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that in later Intel Architecture processors (such as the Pentium Pro processor), locking may occur without the LOCK# signal being asserted. See Intel Architecture Compatibility below.

The LOCK prefix can be prepended only to the following instructions and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Intel Architecture Compatibility

Beginning with the Pentium Pro processor, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism insures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 7 of *Intel Architecture Software Developer's Manual, Volume 3*, the for more information on locking of caches.

Operation

```
AssertLOCK#(DurationOfAccompaningInstruction)
```

Flags Affected

None.

Protected Mode Exceptions

#UD — If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

Real-Address Mode Exceptions

#UD — If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

Virtual-8086 Mode Exceptions

#UD — If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

49.14 LODS/LODSB/LODSW/LODSD—Load String

Opcode	Instruction	Description
AC	LODS m8	Load byte at address DS:(E)SI into AL
AD	LODS m16	Load word at address DS:(E)SI into AX
AD	LODS m32	Load doubleword at address DS:(E)SI into EAX
AC	LODSB	Load byte at address DS:(E)SI into AL
AD	LODSW	Load word at address DS:(E)SI into AX
AD	LODSD	Load doubleword at address DS:(E)SI into EAX

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:EDI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See “REP/REPE/REPZ/REPNE / REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

Operation

```
IF (byte load)
    THEN
        AL ← SRC; (* byte load *)
            THEN IF DF = 0
                THEN (E)SI ← (E)SI + 1;
                ELSE (E)SI ← (E)SI – 1;
            FI;
    ELSE IF (word load)
        THEN
            AX ← SRC; (* word load *)
                THEN IF DF = 0
                    THEN (E)SI ← (E)SI + 2;
                    ELSE (E)SI ← (E)SI – 2;
                FI;
        ELSE (* doubleword transfer *)
            EAX ← SRC; (* doubleword load *)
                THEN IF DF = 0
                    THEN (E)SI ← (E)SI + 4;
                    ELSE (E)SI ← (E)SI – 4;
                FI;
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

49.15 LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Description
E2 *cb*	LOOP *rel8*	Decrement count; jump short if count ≠ 0
E1 *cb*	LOOPE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

Operation

```
IF AddressSize = 32
    THEN
        Count is ECX;
    ELSE (* AddressSize = 16 *)
        Count is CX;
FI;
Count ← Count – 1;

IF instruction is not LOOP
    THEN
        IF (instruction = LOOPE) OR (instruction = LOOPZ)
            THEN
                IF (ZF =1) AND (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
        FI;
        IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
            THEN
                IF (ZF =0 ) AND (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
        FI;
    ELSE (* instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond ← 1;
            ELSE BranchCond ← 0;
        FI;
FI;
IF BranchCond = 1
    THEN
         EIP ← EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN
                EIP ← EIP AND 0000FFFFH;
        FI;
    ELSE
        Terminate loop and continue program execution at EIP;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the offset jumped to is beyond the limits of the code segment.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

49.16 LSL—Load Segment Limit

Opcode	Instruction	Description
0F 03 /*r*	LSL *r16,r/m16*	Load: *r16* ← segment limit, selector *r/m16*
0F 03 /*r*	LSL *r32,r/m32*	Load: *r32* ← segment limit, selector *r/m32*)

Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	No
5	16-bit/32-bit task gate	No
6	16-bit interrupt gate	No
7	16-bit trap gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	No
D	Reserved	No
E	32-bit interrupt gate	No
F	32-bit trap gate	No

Operation

```
IF SRC(Offset) > descriptor table limit
    THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
    AND (CPL > DPL) OR (RPL > DPL)
    OR Segment type is not valid for instruction
        THEN
            ZF ← 0
        ELSE
            temp ← SegmentLimit([SRC]);
            IF (G = 1)
                THEN
                    temp ← ShiftLeft(12, temp) OR 00000FFFH;
            FI;
            IF OperandSize = 32
                THEN
                    DEST ← temp;
                ELSE (*OperandSize = 16*)
                    DEST ← temp AND FFFFH;
            FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is cleared to 0.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	The LSL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LSL instruction is not recognized in virtual-8086 mode.

49.17 LSS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS—Load Far Pointer.

49.18 LTR—Load Task Register

Opcode	Instruction	Description
0F 00 /3	LTR *r/m16*	Load *r/m16* into task register

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

Operation

```
IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
    THEN #GP(segment selector);
FI;
Read segment descriptor;
IF segment descriptor is not for an available TSS THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
TSSsegmentDescriptor(busy) ← 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)
TaskRegister(SegmentSelector) ← SRC;
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The LTR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LTR instruction is not recognized in virtual-8086 mode.